The Impact of Full Adder Designs on ALU Performance and Energy Efficiency

Joseph De La Pascua, Natesha Ramdhani

Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816-2362

Abstract—**This paper works to examine the effects of different designs of full adder circuits on the energy consumption and efficiency of the arithmetic logic unit (ALU) of assembly programs in MIPS. The full adder is the base unit of the ALU since it is capable of executing many of the arithmetic and logic functions that are necessary in the MIPS operational set. The different forms of the full adder being compared are the Spin Hall Effect driven Domain Wall Motion adder (SHE-DWM), the Spin-Torque Transfer in Magnetic Tunnel Junctions adder (STT-MTJ), the Racetrack Memory (RTM)-based Non-Volatile Full Adder (NVFA), and the Spin-Torque Transfer (STT)-based NVFA. The program used to test the energy efficiency of the ALUs based on the above full adders uses a code with two user-input integers. The first output will be a hexadecimal representation of the concatenation of the lesser value of each of the eight 4-bit groupings in the 32-bit version of the input integers. The second output is an array of the count of the occurrences of each hexadecimal value in the first output. In the end, it was found that the SHE-DWM adder was the most efficient adder to implement in the ALU because it had the lowest energy consumption value per ALU instruction, which was 0.6 pJ/ALU instruction. With the test case used, it was found that the total energy consumed with the SHE-DWM based adder was 3775.8 pJ.**

Keywords— **Full Adder, Arithmetic Logic Unit (ALU), Energy Consumption, Energy Efficiency**

I. INTRODUCTION

The program is designed to load, store, and modify an array inside of a MIPS program. The first section of the code is to design a function that would get through two separate 32-bit numbers and pick out the lowest of each byte and provide the output from all the lowest bytes between the two numbers. To implement this in the most energy efficient way possible, only one loop was used for this function. The second part of the code was to design a function that would count each byte of the new number and would report the amount of each hexadecimal value and store the count numbers in an array created in the program. To stay as energy efficient as possible, this part of the code includes one loop.

The test inputs of this code are only two numbers that go through each part of the code. The methodology for testing must include test cases that test for shorter, longer, and average length words. The primary test case was the one provided in the project description. To be thorough in testing, multiple different test cases were used. The outputs are both the new numbers after running through the first part of the code, and the array with the counted numbers from the second part of the code.

A. Project Design

As stated in the introduction, the code has two required parts. The first part is the QuadMinMixer function. This separate function takes in two different 32-bit numbers as inputs and then return one number with the lowest byte of each number in each place. For example, if the two input numbers were 0x34D5 and 0x41CF, the resulting output would give 0x31C5. To implement this function with energy consumption in mind, a single loop is needed. In the start of the function, the inputs are to be stored in the $a0 and $a1 registers. Outside of the loop use registers to hold the count of the loop and the masking number 0xF for the purpose of finding the lowest byte. Inside the loop, the code uses the technique of masking to evaluate both numbers. To mask, the code uses a register containing all ones (0xF) and the "and" function to separate one specific byte in our 32-bit numbers. Once one byte is obtained for each number, we can run a simple logic branch to determine which byte holds a smaller number. Once the smaller byte is determined, the code then adds it to the output register. This is done eight times for each of the eight bytes in the input numbers. After the code executes, the output should contain a 32-bit number composed of the lowest bytes between the two inputs.

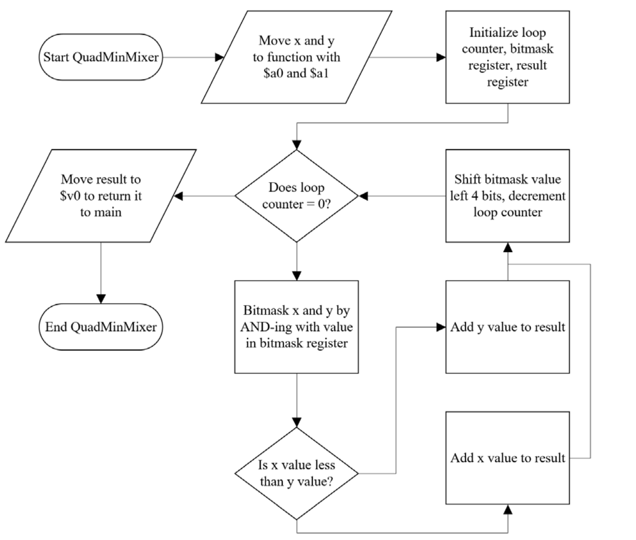

Start QuadMinMixer
Move x and y to function with $a0 and $a1
Initialize loop counter, bitmask register, result register
Move result to $v0 to return it to main
Does loop counter = 0?
Shift bitmask value left 4 bits, decrement loop counter
End QuadMinMixer
Bitmask x and y by AND-ing with value in bitmask register
Add y value to result
Is x value less than y value?
Add x value to result

Fig. 1: Flowchart of QuadMinMixer function in QuadBitCipher assembly program.

The second part of the code contains the number obtained from the first part as the input and an array of counts. The purpose of this code is to count the occurrence of each specific byte and store each of those counts in an array that has the counts for each hexadecimal digit. For example, if our output from part A was 0x31C3, then our count array will have a 1 in the 1's and C's place in the array and a 2 in the 3's spot in the array. There are several ways to do this, but the code implements a more energy and instruction count efficient method than many others. To run the code with as little energy as possible, the array needs to be directly accessed to store and load each byte from the array as recorded. To implement this code, an unrolled loop was used to go through the input number by each byte. To register a single byte from the input number, bit masking was used to isolate each byte. Once isolated from the input number and stored into another register, the code determines the address of that byte in our array and loads the current count number from the array. Once that number is loaded, the number is incremented and stored back into its proper spot in the count array. After the new value is stored in the count array, the final step is to shift the input number to begin to mask the next byte of the number. This loop process will run for each byte in the input. To print the array for the final output, the code has to print the contents of the array separately. To do this, load the syscall code for printing a base ten integer into $v0, load the value at each index into $a0 and then perform a syscall.

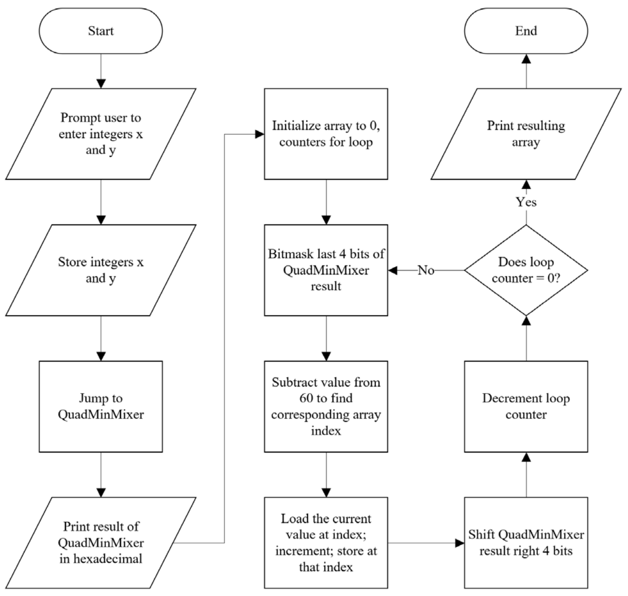
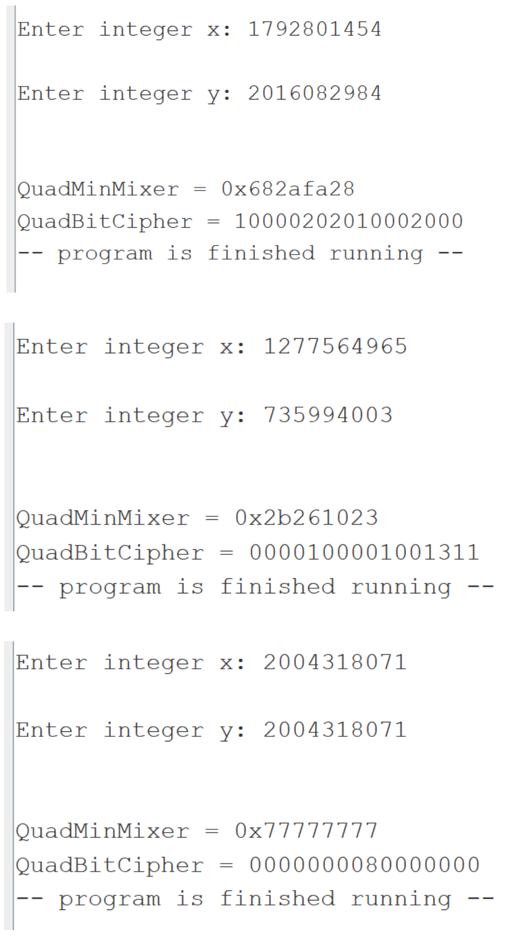

Start
Prompt user to enter integers x and y
Store integers x and y
Jump to QuadMinMixer
Print result of QuadMinMixer in hexadecimal
Initialize array to 0, counters for loop
Bitmask last 4 bits of QuadMinMixer result
Subtract value from 60 to find corresponding array index
Load the current value at index; increment; store at that index
Shift QuadMinMixer result right 4 bits
Decrement loop counter
Does loop counter = 0?
No
Yes
Print resulting array
End

Fig. 2: Flowchart of main section of QuadBitCipher assembly program.

B. *Test Cases*

To test the code thoroughly, several different test cases must be used for specific reasons. The most obvious test cases will be the ones provided in the project description. The first of these test cases uses the 32-bit versions of numbers x=1792801454 and y=2016082984. These two numbers work well because they are both full 32-bit numbers that only take the code about 8 loops for each loop in the code to execute. The second test case provided uses x=1277564965 and y=735994003. Functionally, this test case is similar to the first, but tests the program over a larger range included in the 32-bit versions of these numbers to verify that the program is consistent when the difference in the bit groups is both large and small. One last special case should be tested, when both inputs have the same number. For this test case, the number 0x77777777, or 2004318071 in base 10, will be given as both inputs. The output should be the same number and contain an array that has a count of 8 at the index of the array that corresponds to the 7 digit.


```
Enter integer x: 1792801454

Enter integer y: 2016082984


QuadMinMixer = 0x682afa28
QuadBitCipher = 1000020201000200
-- program is finished running --
```

```
Enter integer x: 1277564965

Enter integer y: 735994003


QuadMinMixer = 0x2b261023
QuadBitCipher = 0000100001001311
-- program is finished running --
```

```
Enter integer x: 2004318071

Enter integer y: 2004318071


QuadMinMixer = 0x77777777
QuadBitCipher = 0000000080000000
-- program is finished running --
```

Fig. 3: Sample outputs of the QuadMinMixer and QuadBitCipher programs using the three test cases described in section I part B.

II. Full-Adder Circuit

A full adder is a circuit that adds three input bits: one bit per place value per number for each of two numbers *a* and *b*, added to one bit carried in c_{in}. The full adder will output two bits to represent the sum s and the value c_{out} carried out after execution of the addition of a, b, and c_{in}.

Full adder cells can be combined to add numbers with more than two bits. A ripple carry adder, for example, cascades full adder cells with one full adder cell representing each bit of the addends and the carry out of the n^{th} full adder cell being the carry in for the $(n+1)^{th}$ full adder cell. Since full adders are able to add numbers, they are also able to subtract numbers because subtraction of two numbers is the equivalent of adding the negative of the second number to the first number. Having the capability to add and subtract numbers makes multiplication and division possible as well, since multiplication is essentially repeated addition and division is essentially repeated subtraction [5]. The truth table of a full adder shows that, in addition to the arithmetic operations of addition, subtraction, multiplication, and division, the full adder can execute Boolean logic operations such as AND, OR, XOR, and XNOR. These functions perform the operation on the *a* and *b* inputs while using the c_{in} input as a control line to switch between AND and OR, and XNOR and XOR. AND and OR are represented on the c_{out} output by a c_{in} value of 1 and 0 respectively. A value of 1 or 0 (respectively) on the c_{in} input will provide the XNOR or XOR operations as the sum output [4]. Since full adders are capable of performing basic arithmetic and logic operations, they are considered to be the bases of the Arithmetic Logic Unit, or ALU [6].

Traditionally, full adders have been comprised of Boolean logic gates. Recently, however, new devices are becoming promising options for the future of full adder implementation. One of these new options is Spin-based Logic-in Memory, which utilizes the torque of the electron spin accumulated on the surface of conductive material, as determined by the Spin-Hall Effect, to drive the motion of the domain wall, and therefore drive the changes in magnitude and sign of the domain wall [1]. Another promising option is based on the "magnetic RAM (MRAM)" seen in magnetic tunnel junctions (MTJs) that make up magnetic adders driving the motion of the domain wall. This MRAM uses "racetrack memory" which is based on the storage of regions of opposing magnetic polarity in nanowires, or "racetracks" [2]. Non-Volatile Full Adders (NVFAs) can utilize Racetrack Memory (RTM) or Spin-Torque transfer (STT) techniques to store their "Logic-In" memory. Because of their unique storage of magnetic polarity on ferromagnetic strips, RTM devices have the benefits of low power and density with high speed. STT devices also use MRAM to store data and perform operations. However, unlike RTM devices, STT devices connect their write circuit directly to the read circuit. This results in possible additional effort needed to isolate read and write devices, but simplified versions of the timing portions and lower addition cycle times in the circuit [3].

III. Results and Discussion

Each of the designs of adders for a single ALU instruction require different amounts of energy to execute. Each of the specific values are located in table I. This value only changes the total energy calculations of the code for the specific logic-based instructions. The memory, branch, jump, and other instruction types are unaffected by the switch in full-adder design. The energy consumption for each instruction is given by

1) ALU = Refer to Table I
2) Branch = 3 pJ
3) Jump = 2 pJ
4) Memory = 100 pJ
5) Other = 5 pJ

Table I: Energy consumption for a single ALU Instruction in the designs provided in [1-3].

Design	Energy Consumption For Each ALU Instruction
SHE-DWM [1]	0.6 pJ
STT-MTJ [2]	6.3 pJ
RTM-based NVFA [3]	1.67 pJ
STT-based NVFA [3]	1.31 pJ

To calculate total energy consumption of the program, the MIPS instruction statistics tool will be used. The test case to calculate this energy total will be the default test case given in the project description. The code has a total dynamic instruction count of 261 instructions, which can be divided into 147 ALU instructions, 10 jump instructions, 17 branch instructions, 32 memory instructions, and 55 other instructions. Based on the instructions for this test case and the amount of energy required for each instruction, the total energy equation for this test is:

$$E_{tot} = (147 * ALU\ Energy) + (10 * 2) + (17 * 3) + (32 * 100) + (55 * 5)$$

Table II: Total energy consumption for the assembly program using designs provided in [1-3].

Design	Total Energy Consumption
SHE-DWM [1]	3634.2 pJ
STT-MTJ [2]	4472.1 pJ
RTM-based NVFA [3]	3791.5 pJ
STT-based NVFA [3]	3738.6 pJ

This energy equation was used to produce the results of table II based off of the individual full adder technologies in table I. From the amount of energy consumed, the SHE-DWM design is the most energy efficient as it completes arithmemtic instructions at only 0.6 pJ/instruction. The least efficient design is the STT-MTJ design with 6.3 pJ per instruction. Although this design does consume more energy, it may use less space or be a faster system.

IV. CONCLUSION

Using each of the designs for the full adder, the code was able to use less energy depending on whichever adder used the least or most energy. Whenever the adder used less energy, the total energy used for the code went down. This caused the most energy efficient system to be the SHE-DWM adder because it used the least amount of energy per instruction. Using the same logic, the most inefficient system was the STT-MTJ based adder because it used the most energy per instruction. This conclusion makes logical since because the total energy is a direct variation with the amount of energy used for one part, with the other parts remaining constant. Technical topics learned from this project include different designs and operations of full adders, manipulating and storing values into MIPS arrays, calculating total energy consumed based from instruction statistics in MIPS, implementation of full adder based ALU designs, and bit masking to isolate an individual byte from a 32-bit number. As stated before, the best design included the SHE-DWM full adder. The total energy consumption of the program with that adder design was 3.6342 nanojoules.

REFERENCES

[1] S. Salehi and R. F. DeMara, "SLIM-ADC: Spin-based Logic-In-Memory Analog to Digital Converter Leveraging SHE-enabled Domain Wall Motion Devices," Microelectronics Journal, Vol. 81, pp. 137–143, November 2018.

[2] H. P. Trinh et al., “Magnetic adder based on racetrack memory,” IEEE Trans. Circuits Syst. I, Reg. Papers, vol. 60, no. 6, pp. 1469–1477, June 2013.

[3] K. Huang, R. Zhao and Y. Lian, "A Low Power and High Sensing Margin Non-Volatile Full Adder Using Racetrack Memory," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 62, no. 4, pp. 1109-1116, April 2015.

[4] B. Lokesh and M. Malathi, "Full adder based reconfigurable spintronic ALU using STT-MTJ," 2013 Annual IEEE India Conference (INDICON), pp. 1-5, December 2013.

[5] Patel and Fung, "Concurrent Error Detection in Multiply and Divide Arrays," in IEEE Transactions on Computers, vol. C-32, no. 4, pp. 417-422, April 1983.

[6] C. H. Chang, J. Gu and M. Zhang, "A review of 0.18-/spl mu/m full adder performances for tree structured arithmetic circuits," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 13, no. 6, pp. 686-695, June 2005.